5. Ichiro

Program Name: Ichiro.java **Input File: ichiro.dat**

Ichiro has been studying multi-dimensioned arrays and writing programs to work with them. He recently found out that not all languages use zero-base arrays. With zero-based arrays, the first column of a 1-D array is element [0], the first row of a 2-D array is row [0], and the first layer of a 3-D array is layer [0]. Some other programming languages treat arrays differently than Java. In Java, a 3-D array of floats with 2 layers, each layer with 3 rows, and each row with 4 columns would be declared as follows:

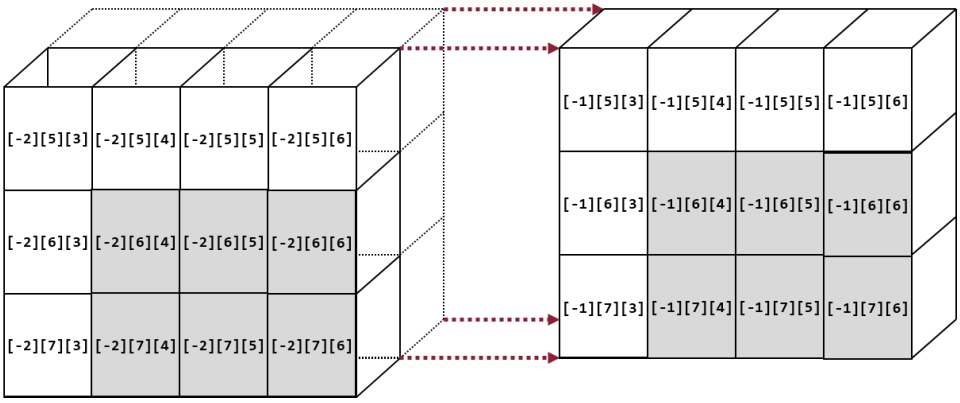
```
float [][][] data = new float[2][3][4];
```

Suppose another language allows the programmer to specify a range for the index values instead of the size of that dimension. Perhaps, something like the following:

```
float [][][] data = new float[-2:-1][5:7][3:6];
```

The arrays are actually the same size but the index values in the program code would have the stated ranges instead of 0..1, 0..2, and 0..3, respectively.

In addition, some programming languages allow for indexing of "slices" of an array. Using the above example, `data[-2:-1][6:7][4:6]` would be a "slice of the above array consisting of the portion of layers `[-2]` and `[-1]` containing only rows `[6]` and `[7]` containing only columns `[4]`, `[5]`, and `[6]`. The "slice" would be 12 cells, those shaded in the following illustration with the "back" layer pushed to the right.



"Slices" of a 3-D array could result in either a 2-D or a 1-D portion of the array like the following examples:

- `data[-2:-2][6:7][4:6]` contains only the shaded cells in the front layer (2-D)
- `data[-1:-1][6:7][6:6]` contains only the shaded cells along the right edge of the back layer (1-D)
- `data[-2:-1][6:6][3:6]` contains only the cells in the second row of both layers (2-D)

Your challenge is to implement such an array and addressing mechanism via Java program code.

Input: First line contains a single integer, number of test cases `T`, which will not exceed 20. For each test case, the next line contains three comma-separated index ranges for a 3-D array, in the form $LB_1:UB_1,LB_2:UB_2,LB_3:UB_3$ where LB_1, LB_2, and LB_3 are lower bounds and UB_1, UB_2, and UB_3 are upper bounds for the layers, rows, and columns, respectively. The bounds will not exceed the range -5000:5000 but there will be no more than 1000 data elements in the array. The following lines will contain comma-separated rows of float data to populate the first layer's first row and then each subsequent line fills the next row, followed by rows of data for subsequent layers. The data is followed by a single line with a single integer `N`, the number of "slices" to be processed, which will not exceed 20. That line is followed by `N` lines, each containing a single comma-separated "slice" in the form $L_1:L_2,R_1:R_2,C_1:C_2$ where `L`'s are layer indexes, `R`'s are row indexes, and `C`'s are column indexes. All index values will be valid for the defined array with the first value ≤ the second value.

Ichiro (cont)

Output: For each test case, output a single line containing the number of elements in the "slice" or "chunk" and the sum of the data items in that "slice" of the array with two decimal places, separated by a single colon **":"** and no additional spacing. Display a line containing 15 equal signs **"==========="** after each test case.

Sample input:

```
2
-2:-1,5:7,3:6
181.79,281.44,403.51,391.91
263.49,378.95,183.85,302.14
142.40,359.64,103.03,370.77
141.24,157.40,377.87,370.56
172.24,104.71,268.41,437.07
283.82,270.07,377.71,380.63
5
-2:-1,6:7,4:6
-2:-2,6:7,4:6
-1:-1,6:7,6:6
-2:-1,5:7,3:6
-1:-1,7:7,5:5
-2:2,216:219,-5:1
942.32,885.71,1422.91,997.57,1641.72,731.13,946.22
975.76,1324.37,654.44,876.39,682.12,998.73,930.60
1652.45,1178.42,1526.27,695.67,1445.09,1652.55,1636.50
717.32,825.24,922.28,683.85,1377.75,542.52,975.56
818.08,987.91,1443.32,1465.33,799.53,1283.15,1038.03
933.23,769.96,1115.08,1522.71,1354.24,1376.36,797.33
869.43,769.70,1432.87,1444.79,1395.87,955.11,834.51
1269.61,880.45,1086.11,515.52,1283.08,1461.40,618.78
864.47,1140.16,876.82,1021.34,1010.39,855.54,942.49
1126.07,674.36,1655.41,563.44,1401.03,821.88,1339.96
718.68,1416.45,1449.02,1580.02,1251.35,607.76,1322.51
969.41,791.67,956.94,1473.29,1537.63,887.91,1107.66
994.01,980.29,1654.21,674.13,760.57,924.61,1326.91
1052.71,1021.48,1016.85,1632.13,1605.99,1195.87,810.72
890.97,1276.26,1540.13,1265.17,984.05,1488.21,1239.79
1021.71,1347.82,1391.01,871.86,1282.98,504.53,941.62
1341.02,1464.87,864.14,1220.91,994.54,757.11,875.26
702.96,1540.39,755.44,1354.18,957.68,1402.33,593.84
1305.26,1343.95,977.22,855.98,561.74,1077.65,1034.47
655.68,1377.09,917.65,1512.59,1524.67,962.17,741.69
6
-1:1,217:218,-4:0
-2:0,216:218,-3:-1
-1:2,217:217,-2:0
0:0,217:219,1:1
-2,2,216:219,-5:1
0:0,218:218,-2:-2
```

Sample output:

```
12:3536.98
6:1698.38
2:817.70
24:6704.65
1:377.71
===============
30:36583.55
27:32724.74
12:15187.84
3:3770.13
140:152095.70
1:1580.02
===============
```